# Software Framework for Building Context-Aware Applications using multiFacet Items

Anca Rarau, Kalman Pusztai, Ioan Salomie
*Computer Science Department, Technical University of Cluj-Napoca*
{anca.rarau, kalman.pusztai, ioan.salomie}@cs.utcluj.ro

## Abstract

*While on the move, the surrounding environment of a mobile application may change quite often. The mobile applications have to be able to properly react to the context changes. The common approach to deal with the application adaptation is based on rules. For each action one must specify the set of rules that triggers the action when the context changes. In this approach the conflictive situations (two or more contradictory actions being simultaneously performed) have to be considered explicitly. The goal of this paper is to present a software framework for building context-aware applications. The adaptation mechanism proposed by the framework successfully manages the conflictive situations without requiring the explicit description of the situations.*

## 1. Introduction

Nowadays people expect their mobile applications to always provide useful information or relevant services within the current context. Location, nearby resources, social environment, etc may give current context. While on the move, the surrounding environment of a mobile application may change quite often. The mobile applications have to be able to properly react to the context changes, i.e. the application has to be context-aware. Therefore while a context-aware application is being built, the application has to be trained how to behave in different contexts.

So far, the development of the context-aware applications, with few exceptions, has been done in an ad-hoc manner. The approach taken by ContextToolkit [5] is worthy of notice as they built a general architecture designed to support context-aware applications. The application consists of *acquire*, *collect*, *transform*, *deliver* and *act-on-context* components. A systematic approach for the development of the context-aware applications is described in [1]. They use conditional rules to describe the behavior of the application in different context. Cooltown [7] resorts to the web programming model. Each real world entity has a web presence represented by a web page. These pages are automatically updated when new information is collected about the entity that it represents. In [2] they use reflection in order to achieve the application adaptation to the context changes.

The common approach to deal with the application adaptation is based on the rules [6, 8, 9]. For each action one must specify the set of the conditions that triggers the action when the context changes. In this approach the conflictive situations, when two or more contradictory actions are simultaneously performed, have to be explicitly considered.

The goal of this paper is to present a software framework for building context-aware applications. The adaptation mechanism proposed by the framework successfully manages the conflictive situations without requiring their explicit description.

The rest of the paper is organized as follows. In Section 2 we discuss the disadvantages of the classical if-then rule approach. In Section 3 we present the multiFacet abstraction. Section 4 presents the proposed software framework. In Section 5 we shortly discuss the infrastructure that deals with the context management. In Section 6 we discuss the performance issues. Section 7 concludes the paper.

## 2. Motivation

The common approach for the application adaptation is the use of either explicit or implicit 'IF condition THEN action' rules. The condition specifies the state of the context. The action leads to a change in the state of an entity. Therefore the if-then rule can be rewritten as follows: 'IF condition THEN change the state of the entity'.

The disadvantage of using classical if-then rules is the lack of implicit support for keeping the state of the entity. It would be useful to memorize the entity state when the condition becomes false and to restore the state next time when the condition is true.
The following set of rules illustrates the situation.

*Rule1: IF tom_in_living*
*THEN light_setting = tom_light_setting*
*Rule 2: IF john_in_living*
*THEN light_seting = john_light_setting*

The rules describe a situation in which the light level is automatically adjusted based on John's or Tom's preferences whenever they enter the living room. What happens if Tom while being in the room changes his light level setting? Is the new level memorized and used next time Tom enters the room? As a matter of fact, the set of rules does not successfully deal with this situation. To overcome this, the set must include rules that capture the moment when someone enters or leaves the room. The previous two rules can be replaced by four rules:

*Rule1: IF tom_enter*
*THEN light_setting = tom_light_setting*
*Rule2: IF tom_exit*
*THEN tom_light_setting = light_setting*
*Rule3: IF john_enter*
*THEN light_setting = john_light_setting*
*Rule4: IF john_exit*
*THEN john_light_setting = light_setting*

What happens if while John is in the room Tom enters the room? What will be the light level? Again, the set of rules do not consider this situation. At least two solutions can be imagined. The luminosity may be adjusted based to John's (first person who enters) preference and it stays like that as long as John is in the room regardless of Tom (second person) presence in the room. Another approach is to use priorities in order to decide upon the light level. E.g. people are given priorities based on their age and luminosity is always adjusted based on the preference of the eldest person in the room. What happens if both John and Tom are in the room and the luminosity is set according to John's wish, but John leaves the room? The set of rules does not cover this particular situation. Therefore we have to improve the set by adding some new rules. Thus the designer of the collection of rules must take into account every possible situation at design time.

Another shortcoming of classical if-then rules is the inability to deal with conflictive rules. In the following example both conditions can be true simultaneously.

*Rule1: IF tom_in_living*
*THEN tv_switch = off*
*Rule2: IF john_in_living*
*THEN tv_switch = on*

The conflictive situation occurs when both John and Tom are in the living room as the actions requested by the two rules are opposite. A possible solution is to add a new rule that explicitly considers the contradiction.

*Rule3: IF tom_in_living and john_in_living*
*THEN tv_switch = off*

This approach becomes more and more intractable when the number of people increases. For the rule designer the situation may become unmanageable as the number of situations that have to be covered by rules increases. It seems that we need some kind of mechanism to place all the rules regarding the tv_switch entity together. This mechanism should be able either to work with priorities assigned to the individual rules or to take into account the first rule whose condition becomes true and neglect the other rules even if their conditions become true.

We continue illustrating the conflictive rules by the following example:

*Rule1: IF tom_in_office*
*THEN music_player_state=play_on*
*Rule2: IF boss_in_tom's_office and working_hours*
*THEN music_player_state=play_off*

Again the two rules can be true at the same time that leads to some on/off opposite actions. In this case, the solution does not imply adding a new rule, but rather assigning priorities to each rule. By assigning the highest priority to the second rule, the player device will be quiet during working hours as long as the boss is in the office regardless of Tom presence.

The above examples clearly show the disadvantages of the classical if-then rule approach. In order to overcome these disadvantages all the rules concerning the same entity must be put together. In this paper we discuss the multiFacet item abstraction that introduces a control mechanism for all the rules regarding an entity.

## 3. multiFacet item abstraction

The context for an application running on a mobile device is changing all the time. The application must adjust to the ever-changing circumstances by exposing and hiding parts of its functionality. An application consists of both components that are context sensitive and components that do not depend on the context. A context sensitive component can be seen as an item with many facets, or a multiFacet item. A facet is aware only of some part of the item functionality. A facet has a condition that behaves like a switch, in the sense that if the condition is true the facet is exposed otherwise the facet is hidden. When a facet is exposed the associated functionality is available to be used by another application or user. When a facet is hidden the associated functionality is not available to be used by another application or user.

The multiFacet item is notified whenever the context changes. The multiFacet item reacts by changing the currently exposed facets. At a given moment, the functionality of the item as a whole is given by the sum of the functionality of every exposed facet. Thus, by the facets exposing and hiding, the functionality of the item goes dynamically richer or poorer.

What happens if the conditions associated to the various facets become true simultaneously? Which of them will be exposed? All facets? Just a part of them? In the following paragraph we describe three exposure strategies.

## 3.1. Exposure strategy

The strategies for exposure specify not only the maximum number of facets that can be exposed at a certain moment, but also which facet(s) is currently exposed. Actually, the exposing strategies specify the behavior of the set of rules in the case two or more rules have their condition true. First option (exclusive strategy) is to trigger the oldest rule having the condition true. Second option (priority based exclusive strategy) is to trigger the highest priority rule having the condition true. Third option (non-exclusive strategy) is to trigger any rule whenever its condition becomes true.
Table 1 summarizes the features of the exposing strategies.

**3.1.1. Exclusive strategy.** Exclusive strategy allows at most one facet to be exposed at a given moment. If, while the facet is exposed, the conditions for some other facets become true, these facets will not be exposed. When the condition of the exposed facet becomes false, it verifies if there are some other facets to be exposed (i.e. with true conditions). If there is just one facet, the facet will be exposed. If there are two or more facets the one whose condition became true first will be exposed.

Below is an example for the exclusive strategy. Imagine an intelligent room having a central module able to detect the occupants of the room and to

Table 1. Features of the exposing strategies

| Strategy | No of maximum exposed facets | What facet is exposed? |
|---|---|---|
| Exclusive | 1 | Oldest facet having true condition. |
| Priority-based exclusive | 1 | Highest priority facet having true condition. In case there are many the oldest one is taken. |
| Non-exclusive | all | Any facet having true condition. |

accordingly adjust the light and temperature to increase their comfort. Let John and Tom be the two persons the room is aware about. The central module can be designed as an exclusive strategy 2-facet item. One facet ($F_J$) knows John's preferences while the other one ($F_T$) knows Tom's light and temperature preferences. If Tom enters the room first, the 2-facet item will expose $F_T$ so the environmental conditions will be set according to Tom's wish. Tom can modify the environmental conditions and the intelligent room will store them for later use. If John enters the room while Tom is in the room the setting does not change because the exclusive strategy has been chosen at design time. If Tom leaves the room, $F_T$ is hidden and $F_J$ is exposed so the temperature and light values are changed to fit John's wish.

**3.1.2. Priority based exclusive strategy.** Priority based exclusive strategy allows at most one facet to be exposed at a given time, namely the highest priority facet having the condition true. If, while a facet is exposed, the condition for another facet becomes true, the priority of the latter facet is verified. If the priority is higher than the priority of the currently exposed facet, then the current facet will be hidden while the latter facet will be exposed. When the condition of the exposed facet becomes false, it verifies if there are some other facets to be exposed (i.e. with true conditions). If there is just one facet, that facet will be exposed. If there are more facets the one with the highest priority is chosen to be exposed. If there is more then one facet with the highest priority, the one whose condition became true first will be exposed.

Here is an example for priority based exclusive strategy. A service that controls how a music player works can be modeled using a priority based exclusive strategy 2-facet item. For the sake of simplicity, we assume that the music player has only the two basic functions: play on and play off. One facet $T_{on}$ exposes and automatically triggers 'play on' function when Tom is in the office. The other facet $T_{off}$ exposes and automatically triggers 'play off' when during the

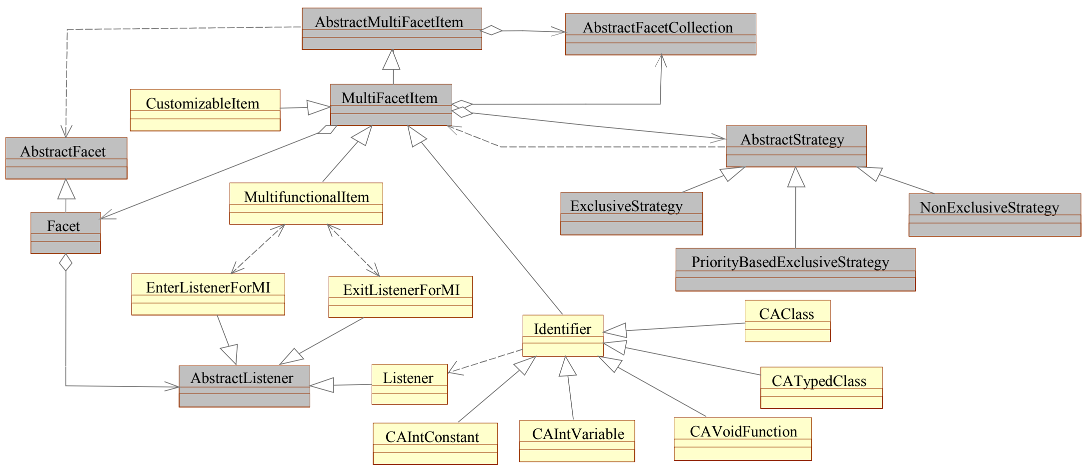

Figure 1. UML class diagram of the framework and its instances

working hours the boss is in the office. The conflictive situation that occurs when both facets should be exposed is successfully overcome by assigning a higher priority to $T_{off}$.

**3.1.3. Non exclusive strategy.** Non exclusive strategy permits any number of facets to be exposed at a given moment. All facets having true conditions are exposed.

Here is an example for non exclusive strategy. Sometimes it is useful for the participants to a meeting, to get a warning before meeting starts. More and more people carry with them mobile devices on which a warning service can be easily installed. The warning service takes the time when the meeting starts from a scheduler application and produces a sound alarm 10 minutes before starting off the meeting. If the level of the noise in the environment is high, the level of the sound alarm also increases. If the level of noise is very high, a vibration alarm will also be triggered. We have designed the warning system as a 3-facet non exclusive strategy item having the following facets:

*$F_1$: IF noise_level < high*
*THEN set the alarm at normal level*
*$F_2$: IF noise_level ≥ high*
*THEN set the alarm at high level*
*$F_3$: IF noise_level > very_high*
*THEN enhance the alarm by vibration*

In case the level of noise overtakes the high level, both $F_2$ and $F_3$ facets are exposed. The alarm will be set at high level and will be enhanced by vibration.

## 4. Software framework for context-aware applications

So far, the development of the context-aware applications mostly has been done in an ad-hoc manner. A notable exception is ContextToolkit [5]. An ad-hoc manner comes with an important disadvantage: the lack of reusability. In order to help the field of the context-aware systems become mature, it is compulsory to provide the right tools to the application developers. A software framework is a mechanism that promotes the reuse of the architectural design and code. Consequently, the effort and time involved in the development of the context-aware applications is significantly reduced.

In the previous section we gave the rationale for the need of using the multiFacet item as a mechanism to coordinate all the rules concerning an entity. Having the multiFacet abstraction as a starting point, we put forward a software framework for the development of context-aware applications. This section describes the components of the framework as well as the relationships among them. We also present a collection of various types of multiFacet items that can be successfully used for developing context-aware applications.

### 4.1. Framework components

The abstract classes *AbstractMultiFacetItem* and *AbstractFacet* model the essential elements of the multiFacet abstraction and the relationship between them. These are superclasses for the main components of the framework, *MultiFacetItem* and *Facet*. These main components encapsulate the behaviour that is common for all the multiFacet items. The *ExclusiveStrategy*, *PriorityBasedExclusiveStrategy* and *NonExclusiveStrategy* are concrete classes that implement the three exposure strategies. The selection of the strategy is done when the item is created and cannot be modified later. A notification procedure may be triggered when a facet changes its state from the exposed to hidden or conversely. The abstract class *AbstractListener* is the superclass for all listeners involved in the notification procedure. In figure 1, which illustrates the UML class diagram of the framework and its instances, the classes that belong to the framework are shaded.

## 4.2. Types of multiFacet items

By instantiating the framework one can build various multiFacet items each with its specific behavior. The instantiation process involves one or more of the following steps:

1) extend *MultiFacetItem* and *AbstractListener* classes;
2) make a decision about listeners' availability. The listeners may be available or not to the application developer. The listeners should be made unavailable to the application developer when they hold predefined bevahiour, which does not have to be altered.
3) decide upon the notification granularity. The notification may happen at the item level or the notification may take place at the facet level. The granularity is an intrinsic feature of the item type. Therefore the granularity is not a choice of the application developer who uses the item, but it is a decision of the developer who creates the type of the item by instantiating the framework.
4) decide upon the suitable exposing strategy. For some item types the strategy is an intrinsic feature of the item, while for some other items it can be chosen by the application developer, when the item is created.

One of our research goals has been to provide context-awareness at the programming language level. In order to achieve this goal we took a first step by developing a framework that provide us with context-aware identifiers and objects. The second step is to extend the C# language with context-aware identifiers and objects. As the focus of this paper is to discuss the proposed software framework for developing context-aware applications we will not consider for discussion the extension of the C# language.

Another research goal has been to discover and provide ready-made multiFacet items that can be useful in various context-aware applications. We have not overlooked the situation in which peculiar applications need customizable items.

**4.2.1 multiFacet identifiers: constant, variable, function (*CAIntConstant*, *CAIntVariable* and *CAVoidFunction* classes).** In contrast with a common identifier, a multiFacet identifier holds more than one value/behavior, each defined for a certain context. Thus, given a context, the multiFacet identifier exposes exactly one value/behavior. We could say that given a context, a multiFacet identifier is similar to a common identifier. As soon as the context changes the value/behaviour of the multiFacet identifier also changes in order to accommodate to the new context.

In order to implement the multiFacet identifier we used a multiFacet item having the exclusive strategy as an intrinsic feature.

A *multiFacet constant* is an identifier having a name, a type and a collection of values, one value for each facet. As for any common constant, the value may not be changed by assignment operation. In the following example we declare an integer constant having two facets and two values, 10 and 100, one for each facet.

```
CAIntConstant ctxMaxParkingTime;
ctxMaxParkingTime = new CAIntConstant(0);
ctxMaxParkingTime.Attach(dayCondition, 10);
ctxMaxParkingTime.Attach(nightCondition, 100);
```

The overall value of the constant is given by the value associated to the current exposed facet. If for the current context no facet has been defined, then the overall value takes the default value (0 in the example above) specified at the creation time.

A *multiFacet variable* is an identifier having a name, a type and a collection of values, one for each facet. The values may be changed by assignment operation. In the following example we declare an integer variable having two facets. The initial value for the case johnCondition is true will be 10, while for the case tomCondition is true will be 20.

```
CAIntVariable ctxLightSettings;
ctxLightSettings = new CAIntVariable(0);
ctxLightSettings.Attach(johnCondition, 10);
ctxLightSettings.Attach(tomCondition, 20);
```

In a certain context, the overall value of the variable is given by the value associated with the current exposed facet. If for the current context no facet has been defined, then the overall value initially takes the default value (0 in the example above) specified at the creation time. While a facet is exposed, the overall value may be changed. Therefore, when the current exposed facet gets hidden the value assigned with the facet is updated to the overall value.

For a common function different calls may trigger different behaviours for various lists of the actual parameters. For a *multiFacet function* the behaviour is specified not only by the set of parameters, but also by the current context. A multiFacet function behaves depending of the context, but it has the same signature in every context. To illustrate a multiFacet function we define a contextual alarm. During the afternoon the alarm behaves as a sound alarm, while during the night the alarm behaves as a light alarm. With a 2-facet function we can easily implement this alarm, by linking one function to each facet.

```
CAFunction ctxAlarm;
ctxAlarm = new CAFunction(
                    new VoidDelegate(NoAlarm));
ctxAlarm.Attach(afternoonCondition,
                    new VoidDelegate(Sound));
ctxAlarm.Attach(nightCondition,
                    new VoidDelegate(Light));

void NoAlarm(int p_Time, int p_Strength) { ... }
void Sound(int p_Time, int p_Strength) { ... }
void Light(int p_Time, int p_Strength)   { ... }
```

At a given moment, the overall behavior is given by the function linked to the current exposed facet. If for the current context no facet has been defined, then the overall behavior is given by a default function (NoAlarm in the example above) specified at the creation time.

**4.2.2. multiFacet objects (*CATypedClass* and *CAClass* classes).** Does it make any sense to create multiFacet objects? In order to answer this question we put forward the case of an object having a data member that takes different values in different contexts. This object is responsible for adjusting the subtitles on a screen. If the user is less than 1 meter away from the screen the font size is set to SMALL. If the user is somewhere between 1 meter and 3 meters the font size increases at MEDIUM, while if the distance between the user and the screen is greater than 3 meters but smaller than 5 meters the font size is LARGE. It seems useful to have the object knowing the three font sizes. The object also has a method called *change()* that actually modifies the font size. The method is automatically triggered when the distance to the screen changes. Moreover the method generates a sound warning when the user is less than 1 meter close to the screen. Both the member data and the method depend on the context. The context-aware application developer might easily implement this scenario if a three-facet object was made available.

We have implemented the *multiFacet object* as a container of objects, an object for each facet. Given a context, the overall behavior of the multiFacet object is given by the behavior of the object linked to the current exposed facet.

We have implemented two categories of multiFacet objects: strongly typed and weakly typed. The former means that all the objects attached to the facets have to be of a certain type. The latter means that no type checking is done when an object is linked to a facet.
A default object is mandatory to be specified at the multiFacet object creation. The default object is twofold:

- the set of the default object methods actually gives the set of the multiFacet object methods i.e. it defines the multiFacet object interface
- if for the current context no facet has been defined, the overall behaviour of the multiFacet object is given by the default object

In order to determine the actual method called on a multiFacet item we use a lookup algorithm.

The lookup algorithm for strongly typed objects:

1. check if the default object has the called method. *Yes*: go to step 2. *No*: raise an exception.
2. call the method on the object linked to the current exposed facet

The lookup algorithm for weakly typed objects:

1. check if the default object has the called method. *Yes*: go to step 2. *No*: raise an exception.
2. check if the object linked to the current exposed facet has the called method. Yes: call the method. No: go to step 3
3. call the method on the default object.

A weakly typed multiFacet object is an instance of CAClass library class while a strongly typed multiFacet object is an instance of CATypedClass.
The 3-facet object responsible with the subtitle adjustment will be an instance of CATypedClass, having objects of type FontSize and SpecializedFontSize on the facets.

```
SpecializedFontSize underone =
            new SpecializedFontSize(SMALL, form);
FontSize undertwo = new FontSize(MEDIUM, form);
FontSize underThree = new FontSize(LARGE, form);
CATypedClass subtitle =
            new CATypedClass(typeof(FontSize),
                              new FontSize(form));
subtitle.Attach(oneMeterCondition, underone);
subtitle.Attach(underThreeMetersCondition, underthree);
subtitle.Attach(underFiveMetersCondition, underfive);
```

If the user is around 1 meter away from the screen *underone* gives the subtitle object behavior. If there are more than 1m but less than 3 meters between the user and the screen then *subtitle* object behaves as *underthree*. If the distance is between 3 meters and 5 meters *underfive* specifies the overall behaviour of *subtitle*. Otherwise the *subtitle* behaves as the default object.

The following examples illustrate how to use various multiFacet identifiers and objects.

```
//the usage of ctxMaxParkingTime constant
if(parkingTime > ctxMaxParkingTime) { ... }

//assign a value to ctxLightSettings variable
ctxLightSettings.Assign(25);

//the use of ctxLightSettings variable
int settings = ctxLightSettings + 3;

//call ctxAlarm function with two parameters
ctxAlarm.Call(10, 20);

//call GetString method on subtitle object
subtitle.InvokeMethod("GetString");
```

Our intention was to provide a way of using multiFacet identifiers and objects as close as possible to the way common identifiers and objects are used. Sometimes the gap between the usage of the multiFacet and common identifiers / objects could not be filled entirely. One case we came across was the assignment operator for a multiFacet identifier. We have developed the framework in C# which does not support the overloading of the assignment operator. Therefore we had to come with another solution for assigning a value to a multiFacet variable, i.e. *Assign* method for assigning values to the multiFacet variables. We used the *InvokeMethod* for calling a method on a multiFacet object.

From the programmer point of view, the description of the reaction to the context changes is done only once at the declaration or creation time. Later in the code the identifiers and objects expose and hide their facets automatically, without any explicit action done by the application developer. When the time comes for the multiFacet identifiers and objects to be used, they will be ready with the right facet exposed.

**Notification**

Besides knowing the context in which a multiFacet item runs it is also important to know the moment when the item 'enters' or 'leaves' the context. In order to catch these moments two listeners are attached to every facet. One listener catches the moment when the item enters a context, i.e. when the facet is being exposed. The other listener catches the moment when the item leaves the context, i.e. the facet is being hidden. These listeners may or may not be available to the application developer.

For multiFacet identifiers and objects, we have forbidden the application developer to modify the predefined behavior of the listeners. Though in some situations the application developer has to be able to insert code to be executed on the expose or hide events provided that the code is not able to modify the predefined behaviour of the listeners. Therefore we provide a notification mechanism accessible to the developer. There are two notifiers, one for exposing the facet and the other for hiding the facet. A notifier is a function that returns void and takes no parameter. The notifier is called post event in the following sense: on exposing a facet, first the facet is exposed then the notifier code is called. On hiding a facet, first the facet is hidden then the notifier code is called.

We will consider again the light settings scenario. It would be nice to be able not only to adjust the light according to the people preferences, but also to display the preferences and the people name. For this we resort to the notification mechanism. *EnterShowLightSettings* and *ExitShowLightSettings* are the two notifiers called when a facet is being exposed and hidden respectively.

```
CAIntVariable ctxLightSettings;
Change inNotify =
                    new Change(EnterShowLightSettings);
Change outNotify =
                   new Change(ExitShowLightSettings);
ctxLightSettings =
                   new CAIntVariable(inNotify, outNotify);
ctxLightSettings.Attach("john", johnCondition);
ctxLightSettings.Attach("tom", tomCondition);

public void EnterShowLightSettings()
{
            // display who is in the room:
            // display the light settings
}

public void ExitShowLightSettings()
{
            //display 'none' to indicate that none is
            //in the room
            //display the default light settings
}
```

**4.2.3. Multifunctional item (*MultifunctionalItem* class).** In this paragraph another instance of the framework, called multifunctional item, is presented. For this item a facet can be seen as a gate to a set of services. If the facet is exposed then the gate is opened that makes the services to be available. If the facet is hidden then the gate is closed subsequently the services are unavailable.

We suppose that there is a mechanism for uniquely identifying each service.

Let $S = \{ s_1, s_2, \ldots, s_n \}$ be the set of services provided be the multifunctional item and let $F = \{ F_1, F_2, \ldots, F_m \}$ be the set of its facets.

*Property 1*
Each facet has a non-empty collection of services

$\forall F_i \ \exists \ S_i = \{ s_{i1}, s_{i2}, \dots \}$

where $S_i \subseteq S$, $i \in \{1, ..., m\}$ and $S_i \neq \phi$

*Property 2*
Every service is exposed on at least one facet.

$\forall s_k \ \exists \ F_i$ so that $s_k \in S_i$

where $k \in \{1, \dots, n\}$ and $i \in \{1, \dots, m\}$

*Property 3*
The same service may be exposed on two or more facets.

$\exists \ s_k$ so that $s_k \in S_i$ and $s_k \in S_j$

where $k \in \{1, \dots, n\}$ and $i, j \in \{1, \dots, m\}$ $i \neq j$

*Property 4*
Two facets may not expose the same set of services.

$\nexists \ F_i, F_j$ so that $S_i = S_j$

where $i, j \in \{1, ..., m\}$ and $i \neq j$

**Notification**

The listener mechanism of the multifunctional items is similar to the mechanism used by the multiFacet identifiers. But unlike identifiers, which call the same listener *Listener* both for exposing and hiding a facet, multifunctional items need two listeners. *EnterListenerForMI* is triggered when a facet is exposed, while hide facet event calls *ExitListenerForMI*. These listeners perform an extra step before calling the notifiers. They update the collection of the current exposed services. The notifiers are defined at the item level, bot at the facet level. The developer has the freedom to choose the exposing strategy.

In order to create a useful instance of a software framework the instance usage must be carefully considered. Several master students built a number of applications, which among other items also contain multifunctional items. Reviewing the way they used the items it helped us to better understand their usage. We have reached the conclusion that often the exposing and hiding notifiers are identical. This clearly shows that what really matter is not to catch the 'moment' when a facet turns on/off, but the 'moment' when the collection of current exposed services is updated. Therefore the constructor for the multifunctional item allows specifying an updating notifier in addition to the exposing and hiding notifiers and strategy.

```
public MultifunctionalItem(AbstractStrategy p_Strategy) ;
public MultifunctionalItem(AbstractStrategy p_Strategy,
                           Change p_Enter, Change p_Exit);
public MultifunctionalItem(AbstractStrategy p_Strategy,
                           Change p_Enter, Change p_Exit,
                           Change p_ UpdateExposedServices);
```

There are two options for attaching services to a facet: either a whole collection at once or individual services. We provide APIs to support both options. For the first option, the call not only attaches the services to the facet, but also creates the facet having the context condition provided as first parameter. In case the facet already exists, the call fails.

```
public int Attach(ILogicalCondition p_Condition,
                  AbstractServiceCollection p_Services);
public int Attach(ILogicalCondition p_Condition,
        AbstractServiceCollection p_Services, int p_Priority);
public int Attach(string p_Description,
                  ILogicalCondition p_Condition,
                  AbstractServiceCollection p_Services);
public int Attach(string p_Description,
                  ILogicalCondition p_Condition,
                  AbstractServiceCollection p_Services,
                  int m_Priority);
```

For the second option, attaching individual services, the approach is opposite. The API call does not have the power to create facets, therefore an individual service may be attached only to an existing facet.

```
public int Attach(ILogicalCondition p_Condition,
                  AbstractService p_Service);
```

For the items we have presented, the application developer cannot directly access the listeners, but she may access the notifiers. The notifiers are weaker than listeners in the sense that they cannot change the predefined behavior triggered when a facet turns on/off, but they can add new behaviour to the predefined behaviour.

A service is currently exposed (consequently can be called by the user or other application) if there is at least one currently exposed facet and the service has been attached to that facet. What happens when the facet is being hidden while the service is still in use? The proper behaviour depends at a great extent on the service itself. The solution involves both the framework and the service developer. The service provides a callback that contains customized behavior for stopping an in use service. The callback is called when the facet is being exposed or hidden.

Table 2. Types of items and their features

| Item type | Exposing strategy | Notification granularity | Listener availability |
|---|---|---|---|
| multiFacet identifier | Exclusive | Item level | No |
| multiFacet object | Exclusive | Item level | No |
| Multifunctional item | Exclusive, Priority based exclusive, Non exclusive | Item level | No |
| Customizable item | Exclusive, Priority based exclusive, Non exclusive | Facet level | Yes |

**4.2.4. Customizable item (*CustomizableItem* class).** Some context-aware situations can be dealt with only if the developer holds full control over the multiFacet item. The developer has to be in charge for choosing the exposing strategy while creating the item. Moreover, the developer may design the listeners from the scratch. The listeners having finer granularity may be specified at the facet level, not at the item level. Thus, the API for creating a facet specifies a context condition, an object that will be available when the facet is exposed and two listeners.

```
public int Attach(ILogicalCondition p_Condition,
                  Object p_Value,
                  AbstractListener p_EnterListener,
                  AbstractListener p_ExitListener);
```

**4.2.5. Summary.** The exposing strategy is preset and cannot be changed for the multiFacet identifiers and objects, but it is the developer's choice for both the multifunctional and customizable items. However once the decision on the strategy is made it cannot be changed later. The granularity of the listeners / notificators is either coarse (at the item level) for identificators, objects and multifunctional items or fine (at the facet level) for customizable items.
Table 2 illustrates the relationship between the types of items, the exposing strategies, the notification granularity and the availability of the listeners.

## 5. Infrastructure

The context-aware applications require an infrastructure able to collect and process context related information. The infrastructure disseminates the context information to the subscribing applications while dealing with scalability, security and privacy [4]. The decoupling of application from the context management relieves the application developer from having to know about a specific context format [3].

The applications build with the proposed software framework need a support infrastructure that notifies the multiFacet items when the context changes. Actually, an item gets notified only if one of its context conditions changes. If this is the case, the item is signaled about the modified context condition and the new truth-value. Having the condition, the item identifies the facet and having the truth-value the item knows if the facet is to be exposed or hidden. An application may consist of context-aware as well as non-context-aware components. Various context-aware components may have facets whose conditions are identical. It would be ineffective to check the same condition more than once. We use a publisher-subscriber architecture for building the dissemination mechanism of the messages about context changes. Both the publisher and the subscriber are multiFacet items. The subscribers are those items that have to be notified when the context changes.

While a multiFacet item subscribes to the publisher links are created between item facets and publisher facets. A link connects a facet from the item to the facet of the publisher having the same context condition.

In the current version of our infrastructure the publisher is the dissemination point for the messages about the context changes. When the context changes the publisher reacts by requesting each facet to re-evaluate its truth-value. If the truth-value for a facet F of the publisher has been modified the fact is propagated to every item having a link to F. Thus a context condition is checked just once no matter how many items use it.

## 6. Experiments

Both context-aware components (i.e. multiFacet items) and non-context-aware components can be found in a context-aware application. The latter come with the overhead of the updating the current facet(s) of the items. We measured this overhead using a HP iPAQ rx3700 at 400MHz and 152MB. We made the measurements for three cases: (1) the context-aware components are 1-facet items, (2) the context-aware components are 2-facet items, (3) the context-aware components are 4-facet items. For each case we varied the number of items up to 30 and consider a repository of 30 context messages. The measurements, illustrated in figure 2, show an overhead of around 19ms for the case the collection of the context-aware components contains only one multiFacet item. The overhead goes up to 68ms for the case the collection of context-aware components contains 30 multiFacet items each with 4 facets.

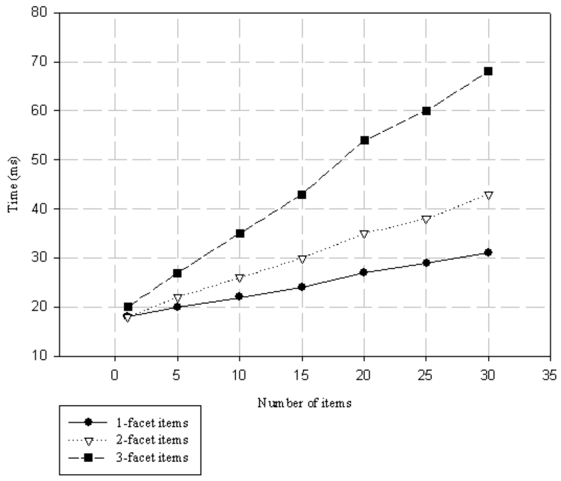

Figure 2. The overhead introduced by the multiFacet items

## 7. Conclusions

As the context-aware systems become more mature, the ad-hoc developing of these systems need to be replaced by systematic approaches. In this paper we presented a reusable approach for creating context-aware applications. The multiFacet abstraction and the software framework build upon the abstraction provide support for the development of the applications able to adapt to the context. The adaptation mechanism proposed by the framework works out the conflictive situations without requiring the explicit description of these situations. Thus, simplifying the task of creating context-aware applications. Those parts of the application that are context-sensitive will be modeled as multiFacet items. We introduced various types of items each with specific behavior.

## 8. References

[1] G. Biegel, V. Cahill, A Framework for Developing Mobile, Context-aware Applications*, 2nd IEEE Conference on Pervasive Computing and Communications*, Orlando, FL, March 14-17

[2] L. Capra, W. Emmerich, C. Mascolo, CARISMA: Context-Aware Reflective mIddleware System for Mobile Applications, *IEEE Transactions on Software Engineering*, Vol. 29, No. 10, October 2003, pp. 929-945

[3] G. Chen, D. Kotz, A Survey of Context-Aware Mobile Computing Research, Technical Report: TR2000-381, Dartmouth College, 2000

[4] G. Chen, D. Kotz, Context Aggregation and Dissemination in Ubiquitous Computing Systems, *4th IEEE Int. Workshop on Mobile Computing Systems and Applications*, 2002

[5] A.K. Dey, Providing Architectural Support for Building Context-Aware Applications, PhD thesis, College of Computing, Georgia Institute of Technology, December 2000

[6] T. Gu, K. Pung, D.Q. Zhang, A service-oriented middleware for building context-aware services, *Journal of Network and Computer Applications*, Volume 28, Issue 1, January 2005, pp.1-18

[7] T. Kindberg, J. Barton, J. Morgan, G. Becker, D. Caswell, P. Debaty, G. Gopal, M. Frid, V. Krishnan, H. Morris, J. Schettino, B. Serra, People, Places, Things: Web Presence for Real World, *IEEE Workshop on Mobile Computing Systems and Applications*, Monterey CA, December 2000

[8] B.N. Schilit, N. Adams, R. Want, Context-Aware Computing Applications, W*orkshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, December 1994, pp. 85-90.

[9] S.S. Yau, F. Karim, An Adaptive Middleware for Context-Sensitive Communications for Real-time Applications in Ubiquitous Computing Environments, *Real-Time Systems*, Volume 26, Issue 1, January 2004, pp. 29-61